Package 'ellmer'

May 7, 2026

Title Chat with Large Language Models

Version 0.4.1

Description Chat with large language models from a range of providers including 'Claude' <https://claude.ai>, 'OpenAI' <https://chatgpt.com>, and more. Supports streaming, asynchronous calls, tool calling, and structured data extraction.

License MIT + file LICENSE

URL https://ellmer.tidyverse.org, https://github.com/tidyverse/ellmer

BugReports https://github.com/tidyverse/ellmer/issues

Depends R (>= 4.1)

Imports cli, coro (>= 1.1.0), glue, httr2 (>= 1.2.1), jsonlite, later (>= 1.4.0), lifecycle, promises (>= 1.5.0), R6, rlang (>= 1.1.0), S7 (>= 0.2.0), tibble, vctrs

Suggests connectcreds, curl (>= 6.0.1), gargle, gitcreds, jose, knitr, magick, openssl, otel (>= 0.2.0), otelsdk (>= 0.2.0), paws.common, png, rmarkdown, shiny, shinychat (>= 0.3.0), testthat (>= 3.0.0), vcr (>= 2.0.0), withr

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate, rmarkdown

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first chat, provider*

Encoding UTF-8

Collate 'utils-S7.R' 'types.R' 'ellmer-package.R' 'tools-def.R' 'content.R' 'provider.R' 'as-json.R' 'batch-chat.R' 'chat-structured.R' 'chat-tools-content.R' 'turns.R' 'chat-tools.R' 'chat-utils.R' 'utils-coro.R' 'chat.R' 'content-image.R' 'content-pdf.R' 'content-replay.R' 'httr2.R' 'import-standalone-defer.R' 'import-standalone-obj-type.R' 'import-standalone-purrr.R' 'import-standalone-types-check.R' 'interpolate.R' 'live.R' 'otel.R' 'parallel-chat.R' 'params.R'

'provider-any.R' 'provider-aws.R'
'provider-openai-compatible.R' 'provider-azure.R'
'provider-claude-files.R' 'provider-claude-tools.R'
'provider-claude.R' 'provider-google.R' 'provider-cloudflare.R'
'provider-databricks.R' 'provider-deepseek.R'
'provider-github.R' 'provider-google-tools.R'
'provider-google-upload.R' 'provider-groq.R'
'provider-huggingface.R' 'provider-lmstudio.R'
'provider-mistral.R' 'provider-ollama.R'
'provider-openai-tools.R' 'provider-openai.R'
'provider-openrouter.R' 'provider-perplexity.R'
'provider-portkey.R' 'provider-snowflake.R' 'provider-vllm.R'
'schema.R' 'stream-controller.R' 'tokens.R' 'tools-built-in.R'
'tools-def-auto.R' 'utils-auth.R' 'utils-callbacks.R'
'utils-cat.R' 'utils-merge.R' 'utils-prettytime.R' 'utils.R'
'zzz.R'

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Hadley Wickham [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4757-117X>),
Joe Cheng [aut],
Aaron Jacobs [aut],
Garrick Aden-Buie [aut] (ORCID:
<https://orcid.org/0000-0002-7111-0077>),
Barret Schloerke [aut] (ORCID: <https://orcid.org/0000-0001-9986-114X>),
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2026-05-07 10:40:02 UTC

Contents

`batch_chat` *Submit multiple chats in one batch*

Description

batch_chat() and batch_chat_structured() currently only work with chat_openai() and chat_anthropic(). They use the OpenAI and Anthropic batch APIs which allow you to submit multiple requests simultaneously. The results can take up to 24 hours to complete, but in return you pay 50% less than usual (but note that ellmer doesn't include this discount in its pricing metadata). If you want to get results back more quickly, or you're working with a different provider, you may want to use parallel_chat() instead.

Since batched requests can take a long time to complete, batch_chat() requires a file path that is used to store information about the batch so you never lose any work. You can either set wait = FALSE or simply interrupt the waiting process, then later, either call batch_chat() to resume where you left off or call batch_chat_completed() to see if the results are ready to retrieve. batch_chat() will store the chat responses in this file, so you can either keep it around to cache the results, or delete it to free up disk space.

This API is marked as experimental since I don't yet know how to handle errors in the most helpful way. Fortunately they don't seem to be common, but if you have ideas, please let me know!

Usage

```
batch_chat(chat, prompts, path, wait = TRUE, ignore_hash = FALSE)

batch_chat_text(chat, prompts, path, wait = TRUE, ignore_hash = FALSE)

batch_chat_structured(
  chat,
  prompts,
  path,
  type,
  wait = TRUE,
  ignore_hash = FALSE,
  convert = TRUE,
  include_tokens = FALSE,
  include_cost = FALSE
)

batch_chat_completed(chat, prompts, path)
```

Arguments

chat	A chat object created by a chat_ function, or a string passed to chat().
prompts	A vector created by interpolate() or a list of character vectors.
path	Path to file (with .json extension) to store state. The file records a hash of the provider, the prompts, and the existing chat turns. If you attempt to reuse the same file with any of these being different, you'll get an error.
wait	If TRUE, will wait for batch to complete. If FALSE, it will return NULL if the batch is not complete, and you can retrieve the results later by re-running batch_chat() when batch_chat_completed() is TRUE.

ignore_hash	If TRUE, will only warn rather than error when the hash doesn't match. You can use this if ellmer has changed the hash structure and you're confident that you're reusing the same inputs.
type	A type specification for the extracted data. Should be created with a type_() function.
convert	If TRUE, automatically convert from JSON lists to R data types using the schema. This typically works best when type is type_object() as this will give you a data frame with one column for each property. If FALSE, returns a list.
include_tokens	If TRUE, and the result is a data frame, will add input_tokens and output_tokens columns giving the total input and output tokens for each prompt.
include_cost	If TRUE, and the result is a data frame, will add cost column giving the cost of each prompt.

Value

For batch_chat(), a list of Chat objects, one for each prompt. For batch_chat_test(), a character vector of text responses. For batch_chat_structured(), a single structured data object with one element for each prompt. Typically, when type is an object, this will will be a data frame with one row for each prompt, and one column for each property.

For any of the aboves, will return NULL if wait = FALSE and the job is not complete.

Examples

```
chat <- chat_openai(model = "gpt-4.1-nano")

# Chat ------------------------------------------------------------------------

prompts <- interpolate("What do people from {{state.name}} bring to a potluck dinner?")
## Not run:
chats <- batch_chat(chat, prompts, path = "potluck.json")
chats

## End(Not run)

# Structured data -------------------------------------------------------------
prompts <- list(
  "I go by Alex. 42 years on this planet and counting.",
  "Pleased to meet you! I'm Jamal, age 27.",
  "They call me Li Wei. Nineteen years young.",
  "Fatima here. Just celebrated my 35th birthday last week.",
  "The name's Robert - 51 years old and proud of it.",
  "Kwame here - just hit the big 5-0 this year."
)
type_person <- type_object(name = type_string(), age = type_number())
## Not run:
data <- batch_chat_structured(
  chat = chat,
  prompts = prompts,
  path = "people-data.json",
  type = type_person
```

```
)
data

## End(Not run)
```

`Chat` *The Chat object*

Description

A `Chat` is a sequence of user and assistant Turns sent to a specific Provider. A `Chat` is a mutable R6 object that takes care of managing the state associated with the chat; i.e. it records the messages that you send to the server, and the messages that you receive back. If you register a tool (i.e. an R function that the assistant can call on your behalf), it also takes care of the tool loop.

You should generally not create this object yourself, but instead call `chat_openai()` or friends instead.

Value

A Chat object

Methods

Public methods:

- `Chat$new()`
- `Chat$get_turns()`
- `Chat$set_turns()`
- `Chat$add_turn()`
- `Chat$get_system_prompt()`
- `Chat$get_model()`
- `Chat$set_system_prompt()`
- `Chat$get_tokens()`
- `Chat$get_cost()`
- `Chat$last_turn()`
- `Chat$chat()`
- `Chat$chat_structured()`
- `Chat$chat_structured_async()`
- `Chat$chat_async()`
- `Chat$stream()`
- `Chat$stream_async()`
- `Chat$register_tool()`
- `Chat$register_tools()`
- `Chat$get_provider()`

- Chat$get_tools()
- Chat$set_tools()
- Chat$on_tool_request()
- Chat$on_tool_result()
- Chat$clone()

`Chat$new()`:

Usage:

```
Chat$new(provider, system_prompt = NULL, echo = "none")
```

Arguments:

`provider` A provider object.

`system_prompt` System prompt to start the conversation with.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `output`: echo text and tool-calling output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method. You can override the default by setting the `ellmer_echo` option.

`Chat$get_turns()`: Retrieve the turns that have been sent and received so far (optionally starting with the system prompt, if any).

Usage:

```
Chat$get_turns(include_system_prompt = FALSE)
```

Arguments:

`include_system_prompt` Whether to include the system prompt in the turns (if any exists).

`Chat$set_turns()`: Replace existing turns with a new list.

Usage:

```
Chat$set_turns(value)
```

Arguments:

`value` A list of Turns.

`Chat$add_turn()`: Add a pair of turns to the chat.

Usage:

```
Chat$add_turn(user, assistant, log_tokens = TRUE)
```

Arguments:

`user` The user Turn.

`assistant` The system Turn.

`log_tokens` Should tokens used in the turn be logged to the session counter?

`Chat$get_system_prompt()`: If set, the system prompt, it not, `NULL`.

Usage:

```
Chat$get_system_prompt()
```

`Chat$get_model()`: Retrieve the model name

Usage:

```
Chat$get_model()
```

`Chat$set_system_prompt()`: Update the system prompt

Usage:

```
Chat$set_system_prompt(value)
```

Arguments:

`value` A character vector giving the new system prompt

`Chat$get_tokens()`: A data frame with token usage and cost data. There are four columns: `input`, `output`, `cached_input`, and `cost`. There is one row for each assistant turn, because token counts and costs are only available when the API returns the assistant's response.

Usage:

```
Chat$get_tokens(include_system_prompt = deprecated())
```

Arguments:

`include_system_prompt` **[Deprecated]**

`Chat$get_cost()`: The cost of this chat

Usage:

```
Chat$get_cost(include = c("all", "last"))
```

Arguments:

`include` The default, `"all"`, gives the total cumulative cost of this chat. Alternatively, use `"last"` to get the cost of just the most recent turn. Incomplete turns (from cancelled or interrupted streams) are excluded because they lack token data.

`Chat$last_turn()`: The last turn returned by the assistant.

Usage:

```
Chat$last_turn(role = c("assistant", "user", "system"))
```

Arguments:

`role` Optionally, specify a role to find the last turn with for the role.

Returns: Either a `Turn` or `NULL`, if no turns with the specified role have occurred.

`Chat$chat()`: Submit input to the chatbot, and return the response as a simple string (probably Markdown).

Usage:

```
Chat$chat(..., echo = NULL)
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images (see `content_image_file()` and `content_image_url()`.

`echo` Whether to emit the response to stdout as it is received. If `NULL`, then the value of `echo` set when the chat object was created will be used.

`Chat$chat_structured()`: Extract structured data

Usage:

```
Chat$chat_structured(..., type, echo = "none", convert = TRUE)
```

Arguments:

`...` The input to send to the chatbot. This is typically the text you want to extract data from, but it can be omitted if the data is obvious from the existing conversation.

`type` A type specification for the extracted data. Should be created with a `type_()` function.

`echo` Whether to emit the response to stdout as it is received. Set to "text" to stream JSON data as it's generated (not supported by all providers).

`convert` Automatically convert from JSON lists to R data types using the schema. For example, this will turn arrays of objects into data frames and arrays of strings into a character vector.

`Chat$chat_structured_async()`: Extract structured data, asynchronously. Returns a promise that resolves to an object matching the type specification.

Usage:

```
Chat$chat_structured_async(..., type, echo = "none", convert = TRUE)
```

Arguments:

`...` The input to send to the chatbot. Will typically include the phrase "extract structured data".

`type` A type specification for the extracted data. Should be created with a `type_()` function.

`echo` Whether to emit the response to stdout as it is received. Set to "text" to stream JSON data as it's generated (not supported by all providers).

`convert` Automatically convert from JSON lists to R data types using the schema. For example, this will turn arrays of objects into data frames and arrays of strings into a character vector.

`Chat$chat_async()`: Submit input to the chatbot, and receive a promise that resolves with the response all at once. Returns a promise that resolves to a string (probably Markdown).

Usage:

```
Chat$chat_async(..., tool_mode = c("concurrent", "sequential"))
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images.

`tool_mode` Whether tools should be invoked one-at-a-time (`"sequential"`) or concurrently (`"concurrent"`). Sequential mode is best for interactive applications, especially when a tool may involve an interactive user interface. Concurrent mode is the default and is best suited for automated scripts or non-interactive applications.

`Chat$stream()`: Submit input to the chatbot, returning streaming results. Returns A coro generator that yields strings. While iterating, the generator will block while waiting for more content from the chatbot.

Usage:

```
Chat$stream(..., stream = c("text", "content"), controller = NULL)
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images.

`stream` Whether the stream should yield only `"text"` or ellmer's rich content types. When `stream = "content"`, `stream()` yields Content objects.

`controller` An optional `stream_controller()` used to cancel the stream from outside the iteration loop.

`Chat$stream_async()`: Submit input to the chatbot, returning asynchronously streaming results. Returns a coro async generator that yields string promises.

Usage:

```
Chat$stream_async(
  ...,
  tool_mode = c("concurrent", "sequential"),
  stream = c("text", "content"),
  controller = NULL
)
```

Arguments:

`...` The input to send to the chatbot. Can be strings or images.

`tool_mode` Whether tools should be invoked one-at-a-time (`"sequential"`) or concurrently (`"concurrent"`). Sequential mode is best for interactive applications, especially when a tool may involve an interactive user interface. Concurrent mode is the default and is best suited for automated scripts or non-interactive applications.

`stream` Whether the stream should yield only `"text"` or ellmer's rich content types. When `stream = "content"`, `stream()` yields Content objects.

`controller` An optional `stream_controller()` used to cancel the stream from outside the iteration loop.

`Chat$register_tool()`: Register a tool (an R function) that the chatbot can use. Learn more in `vignette("tool-calling")`.

Usage:

```
Chat$register_tool(tool)
```

Arguments:

`tool` A tool definition created by `tool()`.

`Chat$register_tools()`: Register a list of tools. Learn more in `vignette("tool-calling")`.

Usage:

```
Chat$register_tools(tools)
```

Arguments:

`tools` A list of tool definitions created by `tool()`.

`Chat$get_provider()`: Get the underlying provider object. For expert use only.

Usage:

```
Chat$get_provider()
```

`Chat$get_tools()`: Retrieve the list of registered tools.

Usage:

```
Chat$get_tools()
```

`Chat$set_tools()`: Sets the available tools. For expert use only; most users should use `register_tool()`.

Usage:

```
Chat$set_tools(tools)
```

Arguments:

`tools` A list of tool definitions created with `tool()`.

`Chat$on_tool_request()`: Register a callback for a tool request event.

Usage:

```
Chat$on_tool_request(callback)
```

Arguments:

`callback` A function to be called when a tool request event occurs, which must have `request` as its only argument.

Returns: A function that can be called to remove the callback.

`Chat$on_tool_result()`: Register a callback for a tool result event.

Usage:

```
Chat$on_tool_result(callback)
```

Arguments:

`callback` A function to be called when a tool result event occurs, which must have `result` as its only argument.

Returns: A function that can be called to remove the callback.

`Chat$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Chat$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
chat <- chat_openai()
chat$chat("Tell me a funny joke")
```

chat *Chat with any provider*

Description

This is a generic interface to all the other `chat_` functions that allow to you pick the provider and the model with a simple string.

Usage

```
chat(
  name,
  ...,
  system_prompt = NULL,
  params = NULL,
  echo = c("none", "output", "all")
)
```

Arguments

`name`	Provider (and optionally model) name in the form `"provider/model"` or `"provider"` (which will use the default model for that provider).
`...`	Arguments passed to the provider function.
`system_prompt`	A system prompt to set the behavior of the assistant.
`params`	Common model parameters, usually created by `params()`.
`echo`	One of the following options: • `none`: don't emit any output (default when running in a function). • `output`: echo text and tool-calling output as it streams in (default when running at the console). • `all`: echo all input and output. Note this only affects the `chat()` method.

chat_anthropic *Chat with an Anthropic Claude model*

Description

Anthropic provides a number of chat based models under the Claude moniker. Note that a Claude Pro membership does not give you the ability to call models via the API; instead, you will need to sign up (and pay for) a developer account.

Usage

```
chat_anthropic(
  system_prompt = NULL,
  params = NULL,
  model = NULL,
  cache = c("5m", "1h", "none"),
  api_args = list(),
  base_url = "https://api.anthropic.com/v1",
  beta_headers = character(),
  api_key = NULL,
  credentials = NULL,
  api_headers = character(),
  echo = NULL
)

chat_claude(
  system_prompt = NULL,
  params = NULL,
  model = NULL,
  cache = c("5m", "1h", "none"),
  api_args = list(),
  base_url = "https://api.anthropic.com/v1",
  beta_headers = character(),
  api_key = NULL,
  credentials = NULL,
  api_headers = character(),
  echo = NULL
)

models_claude(
  base_url = "https://api.anthropic.com/v1",
  api_key = NULL,
  credentials = NULL
)

models_anthropic(
  base_url = "https://api.anthropic.com/v1",
  api_key = NULL,
  credentials = NULL
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
params	Common model parameters, usually created by params().
model	The model to use for the chat (defaults to "claude-sonnet-4-5-20250929"). We regularly update the default, so we strongly recommend explicitly specifying a

model for anything other than casual use. Use `models_anthropic()` to see all options.

`cache` How long to cache inputs? Defaults to "5m" (five minutes). Set to "none" to disable caching or "1h" to cache for one hour.

See details below.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`base_url` The base URL to the endpoint; the default is Claude's public API.

`beta_headers` Optionally, a character vector of beta headers to opt-in claude features that are still in beta.

`api_key` **[Deprecated]** Use `credentials` instead.

`credentials` Override the default credentials. You generally should not need this argument; instead set the `ANTHROPIC_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`.

If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).

`api_headers` Named character vector of arbitrary extra headers appended to every chat API call.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `output`: echo text and tool-calling output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

Caching

Caching with Claude is a bit more complicated than other providers but we believe that on average it will save you both money and time, so we have enabled it by default. With other providers, like OpenAI and Google, you only pay for cache reads, which cost 10% of the normal price. With Claude, you also pay for cache writes, which cost 125% of the normal price for 5 minute caching and 200% of the normal price for 1 hour caching.

How does this affect the total cost of a conversation? Imagine the first turn sends 1000 input tokens and receives 200 output tokens. The second turn must first send both the input and output from the previous turn (1200 tokens). It then sends a further 1000 tokens and receives 200 tokens back.

To compare the prices of these two approaches we can ignore the cost of output tokens, because they are the same for both. How much will the input tokens cost? If we don't use caching, we send 1000 tokens in the first turn and 2200 (1000 + 200 + 1000) tokens in the second turn for a total of 3200 tokens. If we use caching, we'll send (the equivalent of) 1000 * 1.25 = 1250 tokens in the first turn. In the second turn, 1000 of the input tokens will be cached so the total cost is 1000 * 0.1

+ (200 + 1000) * 1.25 = 1600 tokens. That makes a total of 2850 tokens, i.e. 11% fewer tokens, decreasing the overall cost.

Obviously, the details will vary from conversation to conversation, but if you have a large system prompt that you re-use many times you should expect to see larger savings. You can see exactly how many input and cache input tokens each turn uses, along with the total cost, with `chat$get_tokens()`. If you don't see savings for your use case, you can suppress caching with `cache = "none"`.

I know this is already quite complicated, but there's one final wrinkle: Claude will only cache longer prompts, with caching requiring at least 1024-4096 tokens, depending on the model. So don't be surprised it if you don't see any differences with caching if you have a short prompt.

See all the details at https://docs.claude.com/en/docs/build-with-claude/prompt-caching.

See Also

Other chatbots: `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_lmstudio()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openai_compatible()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
chat <- chat_anthropic()
chat$chat("Tell me three jokes about statisticians")
```

---

chat_aws_bedrock *Chat with an AWS bedrock model*

Description

AWS Bedrock provides a number of language models, including those from Anthropic's Claude, using the Bedrock Converse API.

Authentication:

Authentication is handled through {paws.common}, so if authentication does not work for you automatically, you'll need to follow the advice at https://www.paws-r-sdk.com/#credentials. In particular, if your org uses AWS SSO, you'll need to run `aws sso login` at the terminal.

Prompt caching:

Bedrock supports prompt caching via cache checkpoints. When caching is enabled, ellmer places cache checkpoints on the system prompt and the last turn, so that the conversation history is cached across turns.

By default (`cache = "auto"`), caching is enabled for models known to support it (Anthropic Claude and Amazon Nova) and disabled for all other models. You can also set `cache` to `"5m"` or `"1h"` to force a specific TTL, or `"none"` to disable caching entirely. Note that individual models may have minimum input token thresholds before caching takes effect.

Note that `token_usage()` does not currently reflect the cost of writing to the cache, which is priced at a premium over regular input tokens. Cache read savings are reported correctly.

Usage

```
chat_aws_bedrock(
  system_prompt = NULL,
  base_url = NULL,
  model = NULL,
  profile = NULL,
  cache = c("auto", "5m", "1h", "none"),
  params = NULL,
  api_args = list(),
  api_headers = character(),
  echo = NULL
)

models_aws_bedrock(profile = NULL, base_url = NULL)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the API endpoint.
model	The model to use for the chat (defaults to "anthropic.claude-sonnet-4-5-20250929-v1:0"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use `models_models_aws_bedrock()` to see all options. . While ellmer provides a default model, there's no guarantee that you'll have access to it, so you'll need to specify a model that you can. If you're using cross-region inference, you'll need to use the inference profile ID, e.g. `model="us.anthropic.claude-sonnet`
profile	AWS profile to use.
cache	How long to cache inputs? The default, `"auto"`, enables caching with a 5-minute TTL for models known to support it (Anthropic Claude and Amazon Nova) and disables caching for all other models. Set to `"5m"` or `"1h"` to force caching on, or `"none"` to disable it. See details below.
params	Common model parameters, usually created by `params()`.

api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Use `params` for common parameters. Model-specific inference parameters can be provided using the `additionalModelRequestFields` field, for example to enable thinking effort in Anthropic Claude models:
	`api_args = list(`
	`  additionalModelRequestFields = list(`
	`  thinking = list(type = "enabled", budget_tokens = 4000)`
	`  )`
	`)`
	See https://docs.aws.amazon.com/bedrock/latest/userguide/conversation-inference-call.html for more details.

`api_headers` Named character vector of arbitrary extra headers appended to every chat API call.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `output`: echo text and tool-calling output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_lmstudio()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openai_compatible()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
# Basic usage
chat <- chat_aws_bedrock()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_azure_openai *Chat with a model hosted on Azure OpenAI*

Description

The Azure OpenAI server hosts a number of open source models as well as proprietary models from OpenAI.

Built on top of `chat_openai_compatible()`.

Authentication:

`chat_azure_openai()` supports API keys and the `credentials` parameter, but it also makes use of:

- Azure service principals (when the `AZURE_TENANT_ID`, `AZURE_CLIENT_ID`, and `AZURE_CLIENT_SECRET` environment variables are set).
- Interactive Entra ID authentication, like the Azure CLI.
- Viewer-based credentials on Posit Connect. Requires the **connectcreds** package.

Usage

```
chat_azure_openai(
  endpoint = azure_endpoint(),
  model,
  params = NULL,
  api_version = NULL,
  system_prompt = NULL,
  api_key = NULL,
  credentials = NULL,
  api_args = list(),
  echo = c("none", "output", "all"),
  api_headers = character(),
  deployment_id = deprecated()
)
```

Arguments

endpoint	Azure OpenAI endpoint url with protocol and hostname, i.e. `https://{your-resource-name}.openai.` Defaults to using the value of the `AZURE_OPENAI_ENDPOINT` environment variable.
model	The **deployment id** for the model you want to use.
params	Common model parameters, usually created by `params()`.
api_version	The API version to use.
system_prompt	A system prompt to set the behavior of the assistant.
api_key	**[Deprecated]** Use `credentials` instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the `AZURE_OPENAI_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
echo	One of the following options: • `none`: don't emit any output (default when running in a function). • `output`: echo text and tool-calling output as it streams in (default when running at the console). • `all`: echo all input and output. Note this only affects the `chat()` method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.
deployment_id	**[Deprecated]** Use `model` instead.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_azure_openai(model = "gpt-4o-mini")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_cloudflare *Chat with a model hosted on CloudFlare*

Description

Cloudflare Workers AI hosts a variety of open-source AI models. To use the Cloudflare API, you must have an Account ID and an Access Token, which you can obtain by following these instructions.

Built on top of chat_openai_compatible().

Known limitations:

- Tool calling does not appear to work.
- Images don't appear to work.

Usage

```
chat_cloudflare(
  account = cloudflare_account(),
  system_prompt = NULL,
  params = NULL,
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)
```

Arguments

account	The Cloudflare account ID. Taken from the CLOUDFLARE_ACCOUNT_ID env var, if defined.
system_prompt	A system prompt to set the behavior of the assistant.
params	Common model parameters, usually created by params().
api_key	**[Deprecated]** Use credentials instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the CLOUDFLARE_API_KEY environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ(). If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
model	The model to use for the chat (defaults to "meta-llama/Llama-3.3-70b-instruct-fp8-fast"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_cloudflare()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_databricks` *Chat with a model hosted on Databricks*

Description

Databricks provides out-of-the-box access to a number of foundation models and can also serve as a gateway for external models hosted by a third party.

Built on top of `chat_openai_compatible()`.

Authentication:

`chat_databricks()` picks up on ambient Databricks credentials for a subset of the Databricks client unified authentication model. Specifically, it supports:

- Personal access tokens
- Service principals via OAuth (OAuth M2M)
- User account via OAuth (OAuth U2M)
- Authentication via the Databricks CLI
- Posit Workbench-managed credentials
- Viewer-based credentials on Posit Connect. Requires the **connectcreds** package.

Usage

```
chat_databricks(
  workspace = databricks_workspace(),
  system_prompt = NULL,
  model = NULL,
  token = NULL,
  params = NULL,
  api_args = list(),
  echo = c("none", "output", "all"),
  api_headers = character()
)
```

Arguments

`workspace`	The URL of a Databricks workspace, e.g. `"https://example.cloud.databricks.com"`. Will use the value of the environment variable `DATABRICKS_HOST`, if set.
`system_prompt`	A system prompt to set the behavior of the assistant.
`model`	The model to use for the chat (defaults to "databricks-claude-3-7-sonnet"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Available foundational models include: • `databricks-claude-3-7-sonnet` (the default) • `databricks-mixtral-8x7b-instruct` • `databricks-meta-llama-3-1-70b-instruct`

- `databricks-meta-llama-3-1-405b-instruct`

`token`	An authentication token for the Databricks workspace, or `NULL` to use ambient credentials.
`params`	Common model parameters, usually created by `params()`.
`api_args`	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
`echo`	One of the following options: • `none`: don't emit any output (default when running in a function). • `output`: echo text and tool-calling output as it streams in (default when running at the console). • `all`: echo all input and output. Note this only affects the `chat()` method.
`api_headers`	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_lmstudio()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openai_compatible()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_databricks()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_deepseek *Chat with a model hosted on DeepSeek*

Description

Sign up at https://platform.deepseek.com.

Built on top of `chat_openai_compatible()`.

Known limitations:

- Structured data extraction is not supported.
- Images are not supported.

Usage

```
chat_deepseek(
  system_prompt = NULL,
  base_url = "https://api.deepseek.com",
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the endpoint; the default uses DeepSeek.
api_key	**[Deprecated]** Use `credentials` instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the `DEEPSEEK_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
model	The model to use for the chat (defaults to "deepseek-chat"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
params	Common model parameters, usually created by `params()`.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
echo	One of the following options: • `none`: don't emit any output (default when running in a function). • `output`: echo text and tool-calling output as it streams in (default when running at the console). • `all`: echo all input and output. Note this only affects the `chat()` method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_deepseek()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_github *Chat with a model hosted on the GitHub model marketplace*

Description

GitHub Models hosts a number of open source and OpenAI models. To access the GitHub model marketplace, you will need to apply for and be accepted into the beta access program. See https://github.com/marketplace/models for details.

This function is a lightweight wrapper around chat_openai_compatible() with the defaults tweaked for the GitHub Models marketplace.

GitHub also supports the Azure AI Inference SDK, which you can use by setting base_url to "https://models.inference.ai.azure.com/". This endpoint was used in **ellmer** v0.3.0 and earlier.

Usage

```
chat_github(
  system_prompt = NULL,
  base_url = "https://models.github.ai/inference/",
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)

models_github(
  base_url = "https://models.github.ai/",
  api_key = NULL,
  credentials = NULL
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the API endpoint.
api_key	**[Deprecated]** Use credentials instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the GITHUB_PAT environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ(). If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
model	The model to use for the chat (defaults to "gpt-4o"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
params	Common model parameters, usually created by params().
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_github()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

chat_google_gemini *Chat with a Google Gemini or Vertex AI model*

Description

Google's AI offering is broken up into two parts: Gemini and Vertex AI. Most enterprises are likely to use Vertex AI, and individuals are likely to use Gemini.

Use `google_upload()` to upload files (PDFs, images, video, audio, etc.)

Authentication:

These functions try a number of authentication strategies, in this order:

- An API key set in the `GOOGLE_API_KEY` env var, or, for `chat_google_gemini()` only, `GEMINI_API_KEY`.
- Google's default application credentials, if the **gargle** package is installed.
- Viewer-based credentials on Posit Connect, if the **connectcreds** package.
- **[Experimental]**. An browser-based OAuth flow, if you're in an interactive session. This currently uses an unverified OAuth app (so you will get a scary warning); we plan to verify in the near future.

Usage

```
chat_google_gemini(
  system_prompt = NULL,
  base_url = "https://generativelanguage.googleapis.com/v1beta/",
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  api_headers = character(),
  echo = NULL
)

chat_google_vertex(
  location,
  project_id,
  system_prompt = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  api_headers = character(),
  echo = NULL
)

models_google_gemini(
  base_url = "https://generativelanguage.googleapis.com/v1beta/",
  api_key = NULL,
```

```
  credentials = NULL
)

models_google_vertex(location, project_id, credentials = NULL)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the API endpoint.
api_key	**[Deprecated]** Use credentials instead.
credentials	A function that returns a list of authentication headers or NULL, the default, to use ambient credentials. See above for details.
model	The model to use for the chat (defaults to "gemini-2.5-flash"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use models_google_gemini() to see all options.
params	Common model parameters, usually created by params().
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
location	Location, e.g. us-east1, me-central1, africa-south1 or global.
project_id	Project ID.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_google_gemini()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_groq` *Chat with a model hosted on Groq*

Description

Sign up at https://groq.com.

Built on top of `chat_openai_compatible()`.

Usage

```
chat_groq(
  system_prompt = NULL,
  base_url = "https://api.groq.com/openai/v1",
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`base_url`	The base URL to the API endpoint.
`api_key`	**[Deprecated]** Use `credentials` instead.
`credentials`	Override the default credentials. You generally should not need this argument; instead set the `GROQ_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
`model`	The model to use for the chat (defaults to "llama-3.1-8b-instant"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
`params`	Common model parameters, usually created by `params()`.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `output`: echo text and tool-calling output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

`api_headers` Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_huggingface()`, `chat_lmstudio()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openai_compatible()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
chat <- chat_groq()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_huggingface *Chat with a model hosted on Hugging Face Serverless Inference API*

Description

Hugging Face hosts a variety of open-source and proprietary AI models available via their Inference API. To use the Hugging Face API, you must have an Access Token, which you can obtain from your Hugging Face account (ensure that at least "Make calls to Inference Providers" and "Make calls to your Inference Endpoints" is checked).

Built on top of `chat_openai_compatible()`.

Known limitations:

- Some models do not support the chat interface or parts of it, for example `google/gemma-2-2b-it` does not support a system prompt. You will need to carefully choose the model.

Usage

```
chat_huggingface(
  system_prompt = NULL,
  params = NULL,
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
params	Common model parameters, usually created by params().
api_key	**[Deprecated]** Use credentials instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the HUGGINGFACE_API_KEY environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ(). If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
model	The model to use for the chat (defaults to "meta-llama/Llama-3.1-8B-Instruct"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_huggingface()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_lmstudio` *Chat with a local LM Studio model*

Description

To use `chat_lmstudio()` first download and install LM Studio. Then load a model using the LM Studio GUI and start the local server. To learn more about running LM Studio locally, see https://lmstudio.ai/docs/developer/core/server/.

Built on top of `chat_openai_compatible()`.

Usage

```
chat_lmstudio(
  system_prompt = NULL,
  base_url = Sys.getenv("LMSTUDIO_BASE_URL", "http://localhost:1234"),
  model,
  params = NULL,
  api_args = list(),
  echo = NULL,
  credentials = NULL,
  api_headers = character()
)

models_lmstudio(base_url = "http://localhost:1234", credentials = NULL)
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`base_url`	The base URL to the API endpoint.
`model`	The model to use for the chat. Use `models_lmstudio()` to see all options.
`params`	Common model parameters, usually created by `params()`.
`api_args`	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
`echo`	One of the following options: • `none`: don't emit any output (default when running in a function). • `output`: echo text and tool-calling output as it streams in (default when running at the console). • `all`: echo all input and output.

Note this only affects the `chat()` method.

`credentials` LM Studio doesn't require credentials for local usage and in most cases you do not need to provide `credentials`.

However, if you're accessing an LM Studio instance hosted behind a reverse proxy or secured endpoint that enforces bearer-token authentication, you can set the `LMSTUDIO_API_KEY` environment variable or provide a callback function to `credentials`.

`api_headers` Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openai_compatible()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
# https://lmstudio.ai/models/zai-org/glm-4.7-flash
chat <- chat_lmstudio(model = "zai-org/glm-4.7-flash")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_mistral *Chat with a model hosted on Mistral's La Platforme*

Description

Get your API key from https://console.mistral.ai/api-keys.

Built on top of `chat_openai_compatible()`.

Known limitations:

- Tool calling is unstable.
- Images require a model that supports images.

Usage

```
chat_mistral(
  system_prompt = NULL,
  params = NULL,
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)

models_mistral(api_key = mistral_key())
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
params	Common model parameters, usually created by params().
api_key	**[Deprecated]** Use credentials instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the MISTRAL_API_KEY environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ(). If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
model	The model to use for the chat (defaults to "mistral-large-latest"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_mistral()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_ollama *Chat with a local Ollama model*

Description

To use `chat_ollama()` first download and install Ollama. Then install some models either from the command line (e.g. with `ollama pull llama3.1`) or within R using ollamar (e.g. `ollamar::pull("llama3.1")`).

Built on top of `chat_openai_compatible()`.

Known limitations:

- Tool calling is not supported with streaming (i.e. when `echo` is `"text"` or `"all"`)
- Models can only use 2048 input tokens, and there's no way to get them to use more, except by creating a custom model with a different default.
- Tool calling generally seems quite weak, at least with the models I have tried it with.

Usage

```
chat_ollama(
  system_prompt = NULL,
  base_url = Sys.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
  model,
  params = NULL,
  api_args = list(),
  echo = NULL,
  api_key = NULL,
  credentials = NULL,
  api_headers = character()
)

models_ollama(base_url = "http://localhost:11434", credentials = NULL)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the API endpoint.
model	The model to use for the chat. Use models_ollama() to see all options.
params	Common model parameters, usually created by params().
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_key	**[Deprecated]** Use credentials instead.
credentials	Ollama doesn't require credentials for local usage and in most cases you do not need to provide credentials. However, if you're accessing an Ollama instance hosted behind a reverse proxy or secured endpoint that enforces bearer-token authentication, you can set the OLLAMA_API_KEY environment variable or provide a callback function to credentials.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_ollama(model = "llama3.2")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

chat_openai *Chat with an OpenAI model*

Description

This is the main interface to OpenAI's models, using the **responses API**. You can use this to access OpenAI's latest models and features like image generation and web search. If you need to use an OpenAI-compatible API from another provider, or the **chat completions** API with OpenAI,use `chat_openai_compatible()` instead.

Note that a ChatGPT Plus membership does not grant access to the API. You will need to sign up for a developer account (and pay for it) at the developer platform.

Usage

```
chat_openai(
  system_prompt = NULL,
  base_url = "https://api.openai.com/v1",
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  api_headers = character(),
  service_tier = c("auto", "default", "flex", "priority"),
  echo = c("none", "output", "all")
)

models_openai(
  base_url = "https://api.openai.com/v1",
  api_key = NULL,
  credentials = NULL
)
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`base_url`	The base URL to the API endpoint.
`api_key`	**[Deprecated]** Use `credentials` instead.
`credentials`	Override the default credentials. You generally should not need this argument; instead set the `OPENAI_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).

`model` The model to use for the chat (defaults to "gpt-4.1"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use. Use `models_openai()` to see all options.

`params` Common model parameters, usually created by `params()`.

`api_args` Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`api_headers` Named character vector of arbitrary extra headers appended to every chat API call.

`service_tier` Request a specific service tier. There are four options:

- `"auto"` (default): uses the service tier configured in Project settings.
- `"default"`: standard pricing and performance.
- `"flex"`: slower and cheaper.
- `"priority"`: faster and more expensive.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `output`: echo text and tool-calling output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_lmstudio()`, `chat_mistral()`, `chat_ollama()`, `chat_openai_compatible()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
chat <- chat_openai()
chat$chat("
  What is the difference between a tibble and a data frame?
  Answer with a bulleted list
")

chat$chat("Tell me three funny jokes about statisticians")
```

`chat_openai_compatible` *Chat with an OpenAI-compatible model*

Description

This function is for use with OpenAI-compatible APIs, also known as the **chat completions** API. If you want to use OpenAI itself, we recommend `chat_openai()`, which uses the newer **responses** API.

Many providers offer OpenAI-compatible APIs, including:

- Ollama for local models
- vLLM for self-hosted models
- Various cloud providers with OpenAI-compatible endpoints

Usage

```
chat_openai_compatible(
  base_url,
  name = "OpenAI-compatible",
  system_prompt = NULL,
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  api_headers = character(),
  preserve_thinking = FALSE,
  echo = c("none", "output", "all")
)
```

Arguments

`base_url`	The base URL to the endpoint. This parameter is **required** since there is no default for OpenAI-compatible APIs.
`name`	The name of the provider; this is shown in `token_usage()` and is used to compute costs.
`system_prompt`	A system prompt to set the behavior of the assistant.
`api_key`	**[Deprecated]** Use `credentials` instead.
`credentials`	Credentials to use for authentication. If not provided, will attempt to use the `OPENAI_API_KEY` environment variable.
`model`	The model to use for chat. No default; depends on your provider.
`params`	Common model parameters, usually created by `params()`.
`api_args`	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.

`api_headers` Named character vector of arbitrary extra headers appended to every chat API call.

`preserve_thinking`
If `TRUE`, reasoning content returned by the model is included when sending conversation history back to the API. If `FALSE` (the default), reasoning content is still captured in the turn but dropped from subsequent requests. Set to `TRUE` if your provider requires or benefits from seeing prior reasoning in multi-turn conversations.

`echo` One of the following options:

- `none`: don't emit any output (default when running in a function).
- `output`: echo text and tool-calling output as it streams in (default when running at the console).
- `all`: echo all input and output.

Note this only affects the `chat()` method.

Value

A Chat object.

See Also

Other chatbots: `chat_anthropic()`, `chat_aws_bedrock()`, `chat_azure_openai()`, `chat_cloudflare()`, `chat_databricks()`, `chat_deepseek()`, `chat_github()`, `chat_google_gemini()`, `chat_groq()`, `chat_huggingface()`, `chat_lmstudio()`, `chat_mistral()`, `chat_ollama()`, `chat_openai()`, `chat_openrouter()`, `chat_perplexity()`, `chat_portkey()`

Examples

```
## Not run:
# Example with Ollama (requires Ollama running locally)
chat <- chat_openai_compatible(
  base_url = "http://localhost:11434/v1",
  model = "llama2"
)
chat$chat("What is the difference between a tibble and a data frame?")

## End(Not run)
```

---

chat_openrouter *Chat with one of the many models hosted on OpenRouter*

Description

Sign up at https://openrouter.ai.

Support for features depends on the underlying model that you use; see https://openrouter.ai/models for details.

Usage

```
chat_openrouter(
  system_prompt = NULL,
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = c("none", "output", "all"),
  api_headers = character()
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
api_key	**[Deprecated]** Use credentials instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the OPENROUTER_API_KEY environment variable. The best place to set this is in .Renviron, which you can easily edit by calling usethis::edit_r_environ(). If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
model	The model to use for the chat (defaults to "gpt-4o"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
params	Common model parameters, usually created by params().
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_perplexity(), chat_portkey()

Examples

```
## Not run:
chat <- chat_openrouter()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`chat_perplexity` *Chat with a model hosted on perplexity.ai*

Description

Sign up at https://www.perplexity.ai.

Perplexity AI is a platform for running LLMs that are capable of searching the web in real-time to help them answer questions with information that may not have been available when the model was trained.

This function is a Uses OpenAI compatible API via `chat_openai_compatible()` with the defaults tweaked for Perplexity AI.

Usage

```
chat_perplexity(
  system_prompt = NULL,
  base_url = "https://api.perplexity.ai/",
  api_key = NULL,
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)
```

Arguments

`system_prompt`	A system prompt to set the behavior of the assistant.
`base_url`	The base URL to the API endpoint.
`api_key`	**[Deprecated]** Use `credentials` instead.
`credentials`	Override the default credentials. You generally should not need this argument; instead set the `PERPLEXITY_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).

model	The model to use for the chat (defaults to "llama-3.1-sonar-small-128k-online"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
params	Common model parameters, usually created by params().
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_portkey()

Examples

```
## Not run:
chat <- chat_perplexity()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_portkey *Chat with a model hosted on PortkeyAI*

Description

PortkeyAI provides an interface (AI Gateway) to connect through its Universal API to a variety of LLMs providers via a single endpoint.

Usage

```
chat_portkey(
  model,
  system_prompt = NULL,
  base_url = "https://api.portkey.ai/v1",
  api_key = NULL,
  credentials = NULL,
  virtual_key = deprecated(),
  params = NULL,
  api_args = list(),
  echo = NULL,
  api_headers = character()
)

models_portkey(base_url = "https://api.portkey.ai/v1", api_key = portkey_key())
```

Arguments

model	The model name, e.g. `@my-provider/my-model`.
system_prompt	A system prompt to set the behavior of the assistant.
base_url	The base URL to the API endpoint.
api_key	**[Deprecated]** Use `credentials` instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the `PORTKEY_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
virtual_key	**[Deprecated]**. Portkey now recommend supplying the model provider (formerly known as the `virtual_key`), in the model name, e.g. `@my-provider/my-model`. See https://portkey.ai/docs/support/upgrade-to-model-catalog for details. For backward compatibility, the `PORTKEY_VIRTUAL_KEY` env var is still used if the model doesn't include a provider.
params	Common model parameters, usually created by `params()`.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
echo	One of the following options: • `none`: don't emit any output (default when running in a function). • `output`: echo text and tool-calling output as it streams in (default when running at the console). • `all`: echo all input and output. Note this only affects the `chat()` method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

See Also

Other chatbots: chat_anthropic(), chat_aws_bedrock(), chat_azure_openai(), chat_cloudflare(), chat_databricks(), chat_deepseek(), chat_github(), chat_google_gemini(), chat_groq(), chat_huggingface(), chat_lmstudio(), chat_mistral(), chat_ollama(), chat_openai(), chat_openai_compatible(), chat_openrouter(), chat_perplexity()

Examples

```
## Not run:
chat <- chat_portkey()
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

---

chat_snowflake *Chat with a model hosted on Snowflake*

Description

The Snowflake provider allows you to interact with LLM models available through the Cortex LLM REST API.

Authentication:

`chat_snowflake()` picks up the following ambient Snowflake credentials:

- A static OAuth token defined via the `SNOWFLAKE_TOKEN` environment variable.
- Key-pair authentication credentials defined via the `SNOWFLAKE_USER` and `SNOWFLAKE_PRIVATE_KEY` (which can be a PEM-encoded private key or a path to one) environment variables.
- Posit Workbench-managed Snowflake credentials for the corresponding `account`.
- Viewer-based credentials on Posit Connect. Requires the **connectcreds** package.

Known limitations:

Note that Snowflake-hosted models do not support images.

Usage

```
chat_snowflake(
  system_prompt = NULL,
  account = snowflake_account(),
  credentials = NULL,
  model = NULL,
  params = NULL,
  api_args = list(),
  echo = c("none", "output", "all"),
  api_headers = character()
)
```

Arguments

system_prompt	A system prompt to set the behavior of the assistant.
account	A Snowflake account identifier, e.g. "testorg-test_account". Defaults to the value of the SNOWFLAKE_ACCOUNT environment variable.
credentials	A list of authentication headers to pass into httr2::req_headers(), a function that returns them when called, or NULL, the default, to use ambient credentials.
model	The model to use for the chat (defaults to "claude-3-7-sonnet"). We regularly update the default, so we strongly recommend explicitly specifying a model for anything other than casual use.
params	Common model parameters, usually created by params().
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with modifyList().
echo	One of the following options: • none: don't emit any output (default when running in a function). • output: echo text and tool-calling output as it streams in (default when running at the console). • all: echo all input and output. Note this only affects the chat() method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

Examples

```
chat <- chat_snowflake()
chat$chat("Tell me a joke in the form of a SQL query.")
```

---

chat_vllm *Chat with a model hosted by vLLM*

Description

vLLM is an open source library that provides an efficient and convenient LLMs model server. You can use chat_vllm() to connect to endpoints powered by vLLM.

Uses OpenAI compatible API via chat_openai_compatible().

Usage

```
chat_vllm(
  base_url,
  system_prompt = NULL,
  model,
  params = NULL,
  api_args = list(),
  api_key = NULL,
  credentials = NULL,
  echo = NULL,
  api_headers = character()
)

models_vllm(base_url, api_key = NULL, credentials = NULL)
```

Arguments

base_url	The base URL to the API endpoint.
system_prompt	A system prompt to set the behavior of the assistant.
model	The model to use for the chat. Use `models_vllm()` to see all options.
params	Common model parameters, usually created by `params()`.
api_args	Named list of arbitrary extra arguments appended to the body of every chat API call. Combined with the body object generated by ellmer with `modifyList()`.
api_key	**[Deprecated]** Use `credentials` instead.
credentials	Override the default credentials. You generally should not need this argument; instead set the `VLLM_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
echo	One of the following options: • `none`: don't emit any output (default when running in a function). • `output`: echo text and tool-calling output as it streams in (default when running at the console). • `all`: echo all input and output. Note this only affects the `chat()` method.
api_headers	Named character vector of arbitrary extra headers appended to every chat API call.

Value

A Chat object.

Examples

```
## Not run:
chat <- chat_vllm("http://my-vllm.com")
chat$chat("Tell me three jokes about statisticians")

## End(Not run)
```

`claude_file_upload` *Upload, downloand, and manage files for Claude*

Description

[Experimental] Use the beta Files API to upload files to and manage files in Claude. This is currently experimental because the API is in beta and may change. Note that you need `beta-headers = "files-api-2025-04-14"` to use the API.

Claude offers 100GB of file storage per organization, with each file having a maximum size of 500MB. For more details see https://docs.claude.com/en/docs/build-with-claude/files

- `claude_file_upload()` uploads a file and returns an object that you can use in chat.
- `claude_file_list()` lists all uploaded files.
- `claude_file_get()` returns an object for an previously uploaded file.
- `claude_file_download()` downloads the file with the given ID. Note that you can only download files created by skills or the code execution tool.
- `claude_file_delete()` deletes the file with the given ID.

Usage

```
claude_file_upload(
  path,
  base_url = "https://api.anthropic.com/v1/",
  beta_headers = "files-api-2025-04-14",
  credentials = NULL
)

claude_file_list(
  base_url = "https://api.anthropic.com/v1/",
  credentials = NULL,
  beta_headers = "files-api-2025-04-14"
)

claude_file_get(
  file_id,
  base_url = "https://api.anthropic.com/v1/",
  credentials = NULL,
  beta_headers = "files-api-2025-04-14"
)
```

```
claude_file_download(
  file_id,
  path,
  base_url = "https://api.anthropic.com/v1/",
  credentials = NULL,
  beta_headers = "files-api-2025-04-14"
)

claude_file_delete(
  file_id,
  base_url = "https://api.anthropic.com/v1/",
  credentials = NULL,
  beta_headers = "files-api-2025-04-14"
)
```

Arguments

path	Path to download the file to.
base_url	The base URL to the endpoint; the default is Claude's public API.
beta_headers	Beta headers to use for the request. Defaults to `files-api-2025-04-14`.
credentials	Override the default credentials. You generally should not need this argument; instead set the `ANTHROPIC_API_KEY` environment variable. The best place to set this is in `.Renviron`, which you can easily edit by calling `usethis::edit_r_environ()`. If you do need additional control, this argument takes a zero-argument function that returns either a string (the API key), or a named list (added as additional headers to every request).
file_id	ID of the file to get information about, download, or delete.

Examples

```
## Not run:
file <- claude_file_upload("path/to/file.pdf")
chat <- chat_anthropic(beta_headers = "files-api-2025-04-14")
chat$chat("Please summarize the document.", file)

## End(Not run)
```

---

claude_tool_web_fetch *Claude web fetch tool*

Description

Enables Claude to fetch and analyze content from web URLs. Claude can only fetch URLs that appear in the conversation context (user messages or previous tool results). For security reasons, Claude cannot dynamically construct URLs to fetch.

Requires the `web-fetch-2025-09-10` beta header. Learn more in https://docs.claude.com/en/docs/agents-and-tools/tool-use/web-fetch-tool.

Usage

```
claude_tool_web_fetch(
  max_uses = NULL,
  allowed_domains = NULL,
  blocked_domains = NULL,
  citations = FALSE,
  max_content_tokens = NULL
)
```

Arguments

max_uses	Integer. Maximum number of fetches allowed per request.
allowed_domains	Character vector. Restrict fetches to specific domains. Cannot be used with blocked_domains.
blocked_domains	Character vector. Exclude specific domains from fetches. Cannot be used with allowed_domains.
citations	Logical. Whether to include citations in the response. Default is TRUE.
max_content_tokens	Integer. Maximum number of tokens to fetch from each URL.

See Also

Other built-in tools: claude_tool_web_search(), google_tool_web_fetch(), google_tool_web_search(), openai_tool_web_search()

Examples

```
## Not run:
chat <- chat_claude(beta_headers = "web-fetch-2025-09-10")
chat$register_tool(claude_tool_web_fetch())
chat$chat("What are the latest package releases on https://tidyverse.org/blog")

## End(Not run)
```

---

claude_tool_web_search

Claude web search tool

Description

Enables Claude to search the web for up-to-date information. Your organization administrator must enable web search in the Anthropic Console before using this tool, as it costs extra ($10 per 1,000 tokens at time of writing).

Learn more in https://docs.claude.com/en/docs/agents-and-tools/tool-use/web-search-tool.

Usage

```
claude_tool_web_search(
  max_uses = NULL,
  allowed_domains = NULL,
  blocked_domains = NULL,
  user_location = NULL
)
```

Arguments

`max_uses`	Integer. Maximum number of searches allowed per request.
`allowed_domains`	Character vector. Restrict searches to specific domains (e.g., `c("nytimes.com", "bbc.com")`). Cannot be used with `blocked_domains`.
`blocked_domains`	Character vector. Exclude specific domains from searches. Cannot be used with `allowed_domains`.
`user_location`	List with optional elements: `country` (2-letter code), `city`, `region`, and `timezone` (IANA timezone) to localize search results.

See Also

Other built-in tools: `claude_tool_web_fetch()`, `google_tool_web_fetch()`, `google_tool_web_search()`, `openai_tool_web_search()`

Examples

```
## Not run:
chat <- chat_claude()
chat$register_tool(claude_tool_web_search())
chat$chat("What was in the news today?")
chat$chat("What's the biggest news in the economy?")

## End(Not run)
```

`Content` *Content types received from and sent to a chatbot*

Description

Use these functions if you're writing a package that extends ellmer and need to customise methods for various types of content. For normal use, see `content_image_url()` and friends.

ellmer abstracts away differences in the way that different Providers represent various types of content, allowing you to more easily write code that works with any chatbot. This set of classes represents types of content that can be either sent to and received from a provider:

- `ContentText`: simple text (often in markdown format). This is the only type of content that can be streamed live as it's received.
- `ContentImageRemote` and `ContentImageInline`: images, either as a pointer to a remote URL or included inline in the object. See `content_image_file()` and friends for convenient ways to construct these objects.
- `ContentToolRequest`: a request to perform a tool call (sent by the assistant).
- `ContentToolResult`: the result of calling the tool (sent by the user). This object is automatically created from the value returned by calling the `tool()` function. Alternatively, expert users can return a `ContentToolResult` from a `tool()` function to include additional data or to customize the display of the result.

Usage

```
Content()

ContentText(text = stop("Required"))

ContentImage()

ContentImageRemote(url = stop("Required"), detail = "")

ContentImageInline(type = stop("Required"), data = NULL)

ContentToolRequest(
  id = stop("Required"),
  name = stop("Required"),
  arguments = list(),
  tool = NULL,
  extra = list()
)

ContentToolResult(value = NULL, error = NULL, extra = list(), request = NULL)

ContentThinking(thinking = stop("Required"), extra = list())

ContentPDF(
  type = stop("Required"),
  data = stop("Required"),
  filename = stop("Required")
)
```

Arguments

`text`	A single string.
`url`	URL to a remote image.
`detail`	Not currently used.
`type`	MIME type of the image.

`data`	Base64 encoded image data.
`id`	Tool call id (used to associate a request and a result). Automatically managed by **ellmer**.
`name`	Function name
`arguments`	Named list of arguments to call the function with.
`tool`	ellmer automatically matches a tool request to the tools defined for the chatbot. If `NULL`, the request did not match a defined tool.
`extra`	Additional data.
`value`	The results of calling the tool function, if it succeeded.
`error`	The error message, as a string, or the error condition thrown as a result of a failure when calling the tool function. Must be `NULL` when the tool call is successful.
`request`	The ContentToolRequest associated with the tool result, automatically added by **ellmer** when evaluating the tool call.
`thinking`	The text of the thinking output.
`filename`	File name, used to identify the PDF.

Value

S7 objects that all inherit from `Content`

Examples

```
Content()
ContentText("Tell me a joke")
ContentImageRemote("https://www.r-project.org/Rlogo.png")
ContentToolRequest(id = "abc", name = "mean", arguments = list(x = 1:5))
```

---

contents_text *Format contents into a textual representation*

Description

[Experimental]

These generic functions can be use to convert Turn contents or Content objects into textual representations.

- `contents_text()` is the most minimal and only includes ContentText objects in the output.
- `contents_markdown()` returns the text content (which it assumes to be markdown and does not convert it) plus markdown representations of images and other content types.
- `contents_html()` returns the text content, converted from markdown to HTML with `commonmark::markdown_html()`, plus HTML representations of images and other content types.

These content types will continue to grow and change as ellmer evolves to support more providers and as providers add more content types.

Usage

```
contents_text(content, ...)

contents_html(content, ...)

contents_markdown(content, ...)
```

Arguments

content	The Turn or Content object to be converted into text. contents_markdown() also accepts Chat instances to turn the entire conversation history into markdown text.
...	Additional arguments passed to methods.

Value

A string of text, markdown or HTML.

Examples

```
turns <- list(
  UserTurn(list(
    ContentText("What's this image?"),
    content_image_url("https://placehold.co/200x200")
  )),
  AssistantTurn("It's a placeholder image.")
)

lapply(turns, contents_text)
lapply(turns, contents_markdown)
if (rlang::is_installed("commonmark")) {
  contents_html(turns[[1]])
}
```

---

content_image_url *Encode images for chat input*

Description

These functions are used to prepare image URLs and files for input to the chatbot. The content_image_url() function is used to provide a URL to an image, while content_image_file() is used to provide the image data itself.

Usage

```
content_image_url(url, detail = c("auto", "low", "high"))

content_image_file(path, content_type = "auto", resize = "low")

content_image_plot(width = 768, height = 768)
```

Arguments

`url`	The URL of the image to include in the chat input. Can be a `data:` URL or a regular URL. Valid image types are PNG, JPEG, WebP, and non-animated GIF.
`detail`	The detail setting for this image. Can be `"auto"`, `"low"`, or `"high"`.
`path`	The path to the image file to include in the chat input. Valid file extensions are `.png`, `.jpeg`, `.jpg`, `.webp`, and (non-animated) `.gif`.
`content_type`	The content type of the image (e.g. `image/png`). If `"auto"`, the content type is inferred from the file extension.
`resize`	If `"low"`, resize images to fit within 512x512. If `"high"`, resize to fit within 2000x768 or 768x2000. (See the OpenAI docs for more on why these specific sizes are used.) If `"none"`, do not resize. You can also pass a custom string to resize the image to a specific size, e.g. `"200x200"` to resize to 200x200 pixels while preserving aspect ratio. Append `>` to resize only if the image is larger than the specified size, and `!` to ignore aspect ratio (e.g. `"300x200>!"`). All values other than none require the `magick` package.
`width, height`	Width and height in pixels.

Value

An input object suitable for including in the `...` parameter of the `chat()`, `stream()`, `chat_async()`, or `stream_async()` methods.

Examples

```
## Not run:
chat <- chat_openai()
chat$chat(
  "What do you see in these images?",
  content_image_url("https://www.r-project.org/Rlogo.png"),
  content_image_file(system.file("httr2.png", package = "ellmer"))
)

plot(waiting ~ eruptions, data = faithful)
chat <- chat_openai()
chat$chat(
  "Describe this plot in one paragraph, as suitable for inclusion in
   alt-text. You should briefly describe the plot type, the axes, and
   2-5 major visual patterns.",
   content_image_plot()
```

```
)

## End(Not run)
```

`content_pdf_file` *Encode PDFs content for chat input*

Description

These functions are used to prepare PDFs as input to the chatbot. The `content_pdf_url()` function is used to provide a URL to an PDF file, while `content_pdf_file()` is used to for local PDF files.

Not all providers support PDF input, so check the documentation for the provider you are using.

Usage

```
content_pdf_file(path)

content_pdf_url(url)
```

Arguments

`path`, `url`	Path or URL to a PDF file.

Value

A `ContentPDF` object

`create_tool_def` *Create metadata for a tool*

Description

In order to use a function as a tool in a chat, you need to craft the right call to `tool()`. This function helps you do that for documented functions by extracting the function's R documentation and using an LLM to generate the `tool()` call. It's meant to be used interactively while writing your code, not as part of your final code.

If the function has package documentation, that will be used. Otherwise, if the source code of the function can be automatically detected, then the comments immediately preceding the function are used (especially helpful if those are roxygen2 comments). If neither are available, then just the function signature is used.

Note that this function is inherently imperfect. It can't handle all possible R functions, because not all parameters are suitable for use in a tool call (for example, because they're not serializable to simple JSON objects). The documentation might not specify the expected shape of arguments to the level of detail that would allow an exact JSON schema to be generated. Please be sure to review the generated code before using it!

Usage

```
create_tool_def(topic, chat = NULL, echo = interactive(), verbose = FALSE)
```

Arguments

topic	A symbol or string literal naming the function to create metadata for. Can also be an expression of the form pkg::fun.
chat	A Chat object used to generate the output. If NULL (the default) uses chat_openai().
echo	Emit the registration code to the console. Defaults to TRUE in interactive sessions.
verbose	If TRUE, print the input we send to the LLM, which may be useful for debugging unexpectedly poor results.

Value

A register_tool call that you can copy and paste into your code. Returned invisibly if echo is TRUE.

Examples

```
## Not run:
  # These are all equivalent
  create_tool_def(rnorm)
  create_tool_def(stats::rnorm)
  create_tool_def("rnorm")
  create_tool_def("rnorm", chat = chat_azure_openai())

## End(Not run)
```

---

df_schema *Describe the schema of a data frame, suitable for sending to an LLM*

Description

df_schema() gives a column-by-column description of a data frame. For each column, it gives the name, type, label (if present), and number of missing values. For numeric and date/time columns, it also gives the range. For character and factor columns, it also gives the number of unique values, and if there's only a few (<= 10), their values.

The goal is to give the LLM a sense of the structure of the data, so that it can generate useful code, and the output attempts to balance between conciseness and accuracy.

Usage

```
df_schema(df, max_cols = 50)
```

Arguments

df	A data frame to describe.
max_cols	Maximum number of columns to includes. Defaults to 50 to avoid accidentally generating very large prompts.

Examples

```
df_schema(mtcars)
df_schema(iris)
```

---

google_tool_web_fetch *Google URL fetch tool*

Description

When this tool is enabled, you can include URLs directly in your prompts and Gemini will fetch and analyze the content.

Learn more in https://ai.google.dev/gemini-api/docs/url-context.

Usage

```
google_tool_web_fetch()
```

See Also

Other built-in tools: claude_tool_web_fetch(), claude_tool_web_search(), google_tool_web_search(), openai_tool_web_search()

Examples

```
## Not run:
chat <- chat_google_gemini()
chat$register_tool(google_tool_web_fetch())
chat$chat("What are the latest package releases on https://tidyverse.org/blog?")

## End(Not run)
```

google_tool_web_search *Google web search (grounding) tool*

Description

Enables Gemini models to search the web for up-to-date information and ground responses with citations to sources. The model automatically decides when (and how) to search the web based on your prompt. Search results are incorporated into the response with grounding metadata including source URLs and titles.

Learn more in https://ai.google.dev/gemini-api/docs/google-search.

Usage

```
google_tool_web_search()
```

See Also

Other built-in tools: claude_tool_web_fetch(), claude_tool_web_search(), google_tool_web_fetch(), openai_tool_web_search()

Examples

```
## Not run:
chat <- chat_google_gemini()
chat$register_tool(google_tool_web_search())
chat$chat("What was in the news today?")
chat$chat("What's the biggest news in the economy?")

## End(Not run)
```

google_upload *Upload a file to gemini*

Description

[Experimental]

This function uploads a file then waits for Gemini to finish processing it so that you can immediately use it in a prompt. It's experimental because it's currently Gemini specific, and we expect other providers to evolve similar feature in the future.

Uploaded files are automatically deleted after 2 days. Each file must be less than 2 GB and you can upload a total of 20 GB. ellmer doesn't currently provide a way to delete files early; please file an issue if this would be useful for you.

Usage

```
google_upload(
  path,
  base_url = "https://generativelanguage.googleapis.com/",
  api_key = NULL,
  credentials = NULL,
  mime_type = NULL
)
```

Arguments

path	Path to a file to upload.
base_url	The base URL to the API endpoint.
api_key	**[Deprecated]** Use credentials instead.
credentials	A function that returns a list of authentication headers or NULL, the default, to use ambient credentials. See above for details.
mime_type	Optionally, specify the mime type of the file. If not specified, will be guesses from the file extension.

Value

A <ContentUploaded> object that can be passed to $chat().

Examples

```
## Not run:
file <- google_upload("path/to/file.pdf")

chat <- chat_google_gemini()
chat$chat(file, "Give me a three paragraph summary of this PDF")

## End(Not run)
```

---

interpolate *Helpers for interpolating data into prompts*

Description

These functions are lightweight wrappers around glue that make it easier to interpolate dynamic data into a static prompt:

- interpolate() works with a string.
- interpolate_file() works with a file.
- interpolate_package() works with a file in the inst/prompts directory of a package.

Compared to glue, dynamic values should be wrapped in {{ }}, making it easier to include R code and JSON in your prompt.

Usage

```
interpolate(prompt, ..., .envir = parent.frame())

interpolate_file(path, ..., .envir = parent.frame())

interpolate_package(package, path, ..., .envir = parent.frame())
```

Arguments

prompt	A prompt string. You should not generally expose this to the end user, since glue interpolation makes it easy to run arbitrary code.
...	Define additional temporary variables for substitution.
.envir	Environment to evaluate ... expressions in. Used when wrapping in another function. See vignette("wrappers", package = "glue") for more details.
path	A path to a prompt file (often a .md). In interpolate_package(), this path is relative to inst/prompts.
package	Package name.

Value

A {glue} string.

Examples

```
joke <- "You're a cool dude who loves to make jokes. Tell me a joke about {{topic}}."

# You can supply valuese directly:
interpolate(joke, topic = "bananas")

# Or allow interpolate to find them in the current environment:
topic <- "applies"
interpolate(joke)
```

---

live_console *Open a live chat application*

Description

- live_console() lets you chat interactively in the console.
- live_browser() lets you chat interactively in a browser.

Note that these functions will mutate the input chat object as you chat because your turns will be appended to the history.

Usage

```
live_console(chat, quiet = FALSE)

live_browser(chat, quiet = FALSE)
```

Arguments

chat	A chat object created by chat_openai() or friends.
quiet	If TRUE, suppresses the initial message that explains how to use the console.

Value

(Invisibly) The input chat.

Examples

```
## Not run:
chat <- chat_anthropic()
live_console(chat)
live_browser(chat)

## End(Not run)
```

---

openai_tool_web_search

OpenAI web search tool

Description

Enables OpenAI models to search the web for up-to-date information. The search behavior varies by model: non-reasoning models perform simple searches, while reasoning models can perform agentic, iterative searches.

Learn more at https://platform.openai.com/docs/guides/tools-web-search

Usage

```
openai_tool_web_search(
  allowed_domains = NULL,
  user_location = NULL,
  external_web_access = TRUE
)
```

Arguments

`allowed_domains`
: Character vector. Restrict searches to specific domains (e.g., `c("nytimes.com", "bbc.com")`). Maximum 20 domains. URLs will be automatically cleaned (http/https prefixes removed).

`user_location`
: List with optional elements: `country` (2-letter ISO code), `city`, `region`, and `timezone` (IANA timezone) to localize search results.

`external_web_access`
: Logical. Whether to allow live internet access (`TRUE`, default) or use only cached/indexed results (`FALSE`).

See Also

Other built-in tools: `claude_tool_web_fetch()`, `claude_tool_web_search()`, `google_tool_web_fetch()`, `google_tool_web_search()`

Examples

```
## Not run:
chat <- chat_openai()
chat$register_tool(openai_tool_web_search())
chat$chat("Very briefly summarise the top 3 news stories of the day")
chat$chat("Of those stories, which one do you think was the most interesting?")

## End(Not run)
```

`parallel_chat` *Submit multiple chats in parallel*

Description

If you have multiple prompts, you can submit them in parallel. This is typically considerably faster than submitting them in sequence, especially with Gemini and OpenAI.

If you're using `chat_openai()` or `chat_anthropic()` and you're willing to wait longer, you might want to use `batch_chat()` instead, as it comes with a 50% discount in return for taking up to 24 hours.

Usage

```
parallel_chat(
  chat,
  prompts,
  max_active = 10,
  rpm = 500,
  on_error = c("return", "continue", "stop")
)
```

```
parallel_chat_text(
  chat,
  prompts,
  max_active = 10,
  rpm = 500,
  on_error = c("return", "continue", "stop")
)

parallel_chat_structured(
  chat,
  prompts,
  type,
  convert = TRUE,
  include_tokens = FALSE,
  include_cost = FALSE,
  max_active = 10,
  rpm = 500,
  on_error = c("return", "continue", "stop")
)
```

Arguments

chat	A chat object created by a chat_ function, or a string passed to chat().
prompts	A vector created by interpolate() or a list of character vectors.
max_active	The maximum number of simultaneous requests to send. For chat_anthropic(), note that the number of active connections is limited primarily by the output tokens per minute limit (OTPM) which is estimated from the max_tokens parameter, which defaults to 4096. That means if your usage tier limits you to 16,000 OTPM, you should either set max_active = 4 (16,000 / 4096) to decrease the number of active connections or use params() in chat_anthropic() to decrease max_tokens.
rpm	Maximum number of requests per minute.
on_error	What to do when a request fails. One of: • "return" (the default): stop processing new requests, wait for in flight requests to finish, then return. • "continue": keep going, performing every request. • "stop": stop processing and throw an error.
type	A type specification for the extracted data. Should be created with a type_() function.
convert	If TRUE, automatically convert from JSON lists to R data types using the schema. This typically works best when type is type_object() as this will give you a data frame with one column for each property. If FALSE, returns a list.
include_tokens	If TRUE, and the result is a data frame, will add input_tokens and output_tokens columns giving the total input and output tokens for each prompt.
include_cost	If TRUE, and the result is a data frame, will add cost column giving the cost of each prompt.

Value

For `parallel_chat()`, a list with one element for each prompt. Each element is either a Chat object (if successful), a `NULL` (if the request wasn't performed) or an error object (if it failed).

For `parallel_chat_text()`, a character vector with one element for each prompt. Requests that weren't succesful get an `NA`.

For `parallel_chat_structured()`, a single structured data object with one element for each prompt. Typically, when `type` is an object, this will be a tibble with one row for each prompt, and one column for each property. If the output is a data frame, and some requests error, an `.error` column will be added with the error objects.

Examples

```
chat <- chat_openai()

# Chat --------------------------------------------------------------------
country <- c("Canada", "New Zealand", "Jamaica", "United States")
prompts <- interpolate("What's the capital of {{country}}?")
parallel_chat(chat, prompts)

# Structured data ---------------------------------------------------------
prompts <- list(
  "I go by Alex. 42 years on this planet and counting.",
  "Pleased to meet you! I'm Jamal, age 27.",
  "They call me Li Wei. Nineteen years young.",
  "Fatima here. Just celebrated my 35th birthday last week.",
  "The name's Robert - 51 years old and proud of it.",
  "Kwame here - just hit the big 5-0 this year."
)
type_person <- type_object(name = type_string(), age = type_number())
parallel_chat_structured(chat, prompts, type_person)
```

---

params *Standard model parameters*

Description

This helper function makes it easier to create a list of parameters used across many models. The parameter names are automatically standardised and included in the correctly place in the API call.

Note that parameters that are not supported by a given provider will generate a warning, not an error. This allows you to use the same set of parameters across multiple providers.

Usage

```
params(
  temperature = NULL,
  top_p = NULL,
```

```
  top_k = NULL,
  frequency_penalty = NULL,
  presence_penalty = NULL,
  seed = NULL,
  max_tokens = NULL,
  log_probs = NULL,
  stop_sequences = NULL,
  reasoning_effort = NULL,
  reasoning_tokens = NULL,
  ...
)
```

Arguments

`temperature`	Temperature of the sampling distribution.
`top_p`	The cumulative probability for token selection.
`top_k`	The number of highest probability vocabulary tokens to keep.
`frequency_penalty`	Frequency penalty for generated tokens.
`presence_penalty`	Presence penalty for generated tokens.
`seed`	Seed for random number generator.
`max_tokens`	Maximum number of tokens to generate.
`log_probs`	Include the log probabilities in the output?
`stop_sequences`	A character vector of tokens to stop generation on.
`reasoning_effort`, `reasoning_tokens`	How much effort to spend thinking? `ressoning_effort` is a string, like "low", "medium", "high". `reasoning_tokens` is an integer, giving a maximum token budget. Each provider only takes one of these two parameters.
`...`	Additional named parameters to send to the provider.

`Provider` *A chatbot provider*

Description

A Provider captures the details of one chatbot service/API. This captures how the API works, not the details of the underlying large language model. Different providers might offer the same (open source) model behind a different API.

Usage

```
Provider(
  name = stop("Required"),
  model = stop("Required"),
  base_url = stop("Required"),
  params = list(),
  extra_args = list(),
  extra_headers = character(0),
  credentials = function() NULL
)
```

Arguments

`name`	Name of the provider.
`model`	Name of the model.
`base_url`	The base URL for the API.
`params`	A list of standard parameters created by `params()`.
`extra_args`	Arbitrary extra arguments to be included in the request body.
`extra_headers`	Arbitrary extra headers to be added to the request.
`credentials`	A zero-argument function that returns the credentials to use for authentication. Can either return a string, representing an API key, or a named list of headers.

Details

To add support for a new backend, you will need to subclass `Provider` (adding any additional fields that your provider needs) and then implement the various generics that control the behavior of each provider.

Value

An S7 Provider object.

Examples

```
Provider(
  name = "CoolModels",
  model = "my_model",
  base_url = "https://cool-models.com"
)
```

`stream_controller` *Create a stream controller*

Description

Creates a controller that can cancel an in-progress stream. Pass it to Chat's `$stream()` or `$stream_async()` via the `controller` argument, then call `$cancel()` from anywhere (e.g. a Shiny observer) to stop the stream after the next chunk arrives.

The same controller can be reused across multiple streams. Call `$reset()` to clear the cancelled state, or pass it directly to a new `$stream()` call — it will be reset automatically.

Usage

```
stream_controller()
```

Value

An `ellmer_stream_controller` object with the following elements:

- `$cancel(reason = "cancelled")`: Cancel the stream. The `reason` string is stored on the controller and used as the AssistantPartialTurn's `reason` property.
- `$reset()`: Clear the cancelled state and reason.
- `$cancelled`: A logical flag indicating whether the controller has been cancelled.
- `$reason`: The cancellation reason string, or `NULL` if not cancelled.

Async cancellation in Shiny

In a Shiny app, use an ExtendedTask for non-blocking chat and a `stream_controller()` to wire up a cancel button:

```
controller <- stream_controller()

chat_task <- ExtendedTask$new(function(user_query, controller = NULL) {
  chat <- chat_openai(model = "gpt-4.1-nano")
  stream <- chat$stream_async(user_query, controller = controller)
  shinychat::markdown_stream("response", stream)
})

observeEvent(input$ask, {
  controller <<- stream_controller()
  chat_task$invoke(input$query, controller = controller)
})

observeEvent(input$cancel, {
  controller$cancel()
})
```

Examples

```
chat <- chat_openai(model = "gpt-5.4-nano")

ctrl <- stream_controller()
stream <- chat$stream("Write a short story.", controller = ctrl)

i <- 0
coro::loop(for (chunk in stream) {
  i <- i + 1
  if (i > 10) ctrl$cancel()
})

chat
```

`token_usage` *Report on token usage in the current session*

Description

Call this function to find out the cumulative number of tokens that you have sent and recieved in the current session. The price will be shown if known.

Usage

```
token_usage()
```

Value

A data frame

Examples

```
token_usage()
```

`tool` *Define a tool*

Description

Annotate a function for use in tool calls, by providing a name, description, and type definition for the arguments.

Learn more in `vignette("tool-calling")`.

Usage

```
tool(
  fun,
  description,
  ...,
  arguments = list(),
  name = NULL,
  convert = TRUE,
  annotations = list(),
  .name = deprecated(),
  .description = deprecated(),
  .convert = deprecated(),
  .annotations = deprecated()
)
```

Arguments

`fun`	The function to be invoked when the tool is called. The return value of the function is sent back to the chatbot. Expert users can customize the tool result by returning a ContentToolResult object.
`description`	A detailed description of what the function does. Generally, the more information that you can provide here, the better.
`...`	**[Deprecated]** Use `arguments` instead.
`arguments`	A named list that defines the arguments accepted by the function. Each element should be created by a `type_*()` function. Use `type_ignore()` if you don't want the LLM to provide that argument (e.g., because the R function has a suitable default value).
`name`	The name of the function. This can be omitted if `fun` is an existing function (i.e. not defined inline).
`convert`	Should JSON inputs be automatically convert to their R data type equivalents? Defaults to `TRUE`.
`annotations`	Additional properties that describe the tool and its behavior. Usually created by `tool_annotations()`, where you can find a description of the annotation properties recommended by the Model Context Protocol.
`.name`, `.description`, `.convert`, `.annotations`	**[Deprecated]** Please switch to the non-prefixed equivalents.

Value

An S7 `ToolDef` object.

ellmer 0.3.0

In ellmer 0.3.0, the definition of the `tool()` function changed quite a bit. To make it easier to update old versions, you can use an LLM with the following system prompt

```
Help the user convert an ellmer 0.2.0 and earlier tool definition into a
ellmer 0.3.0 tool definition. Here's what changed:

* All arguments, apart from the first, should be named, and the argument
  names no longer use `.` prefixes. The argument order should be function,
  name (as a string), description, then arguments, then anything

* Previously `arguments` was passed as `...`, so all type specifications
  should now be moved into a named list and passed to the `arguments`
  argument. It can be omitted if the function has no arguments.

```R
# old
tool(
  add,
  "Add two numbers together"
  x = type_number(),
  y = type_number()
)

# new
tool(
  add,
  name = "add",
  description = "Add two numbers together",
  arguments = list(
    x = type_number(),
    y = type_number()
  )
)
```

Don't respond; just let the user provide function calls to convert.
```

See Also

Other tool calling helpers: tool_annotations(), tool_reject()

Examples

```
# First define the metadata that the model uses to figure out when to
# call the tool
tool_rnorm <- tool(
  rnorm,
  description = "Draw numbers from a random normal distribution",
  arguments = list(
    n = type_integer("The number of observations. Must be a positive integer."),
    mean = type_number("The mean value of the distribution."),
   sd = type_number("The standard deviation of the distribution. Must be a non-negative number.")
```

```
  )
)
tool_rnorm(n = 5, mean = 0, sd = 1)

chat <- chat_openai()
# Then register it
chat$register_tool(tool_rnorm)

# Then ask a question that needs it.
chat$chat("Give me five numbers from a random normal distribution.")

# Look at the chat history to see how tool calling works:
chat
# Assistant sends a tool request which is evaluated locally and
# results are sent back in a tool result.
```

`tool_annotations` *Tool annotations*

Description

Tool annotations are additional properties that, when passed to the `.annotations` argument of `tool()`, provide additional information about the tool and its behavior. This information can be used for display to users, for example in a Shiny app or another user interface.

The annotations in `tool_annotations()` are drawn from the Model Context Protocol and are considered *hints*. Tool authors should use these annotations to communicate tool properties, but users should note that these annotations are not guaranteed.

Usage

```
tool_annotations(
  title = NULL,
  read_only_hint = NULL,
  open_world_hint = NULL,
  idempotent_hint = NULL,
  destructive_hint = NULL,
  ...
)
```

Arguments

`title`	A human-readable title for the tool.
`read_only_hint`	If `TRUE`, the tool does not modify its environment.
`open_world_hint`	If `TRUE`, the tool may interact with an "open world" of external entities. If `FALSE`, the tool's domain of interaction is closed. For example, the world of a web search tool is open, but the world of a memory tool is not.

idempotent_hint
: If TRUE, calling the tool repeatedly with the same arguments will have no additional effect on its environment. (Only meaningful when read_only_hint is FALSE.)

destructive_hint
: If TRUE, the tool may perform destructive updates to its environment, otherwise it only performs additive updates. (Only meaningful when read_only_hint is FALSE.)

...
: Additional named parameters to include in the tool annotations.

Value

A list of tool annotations.

See Also

Other tool calling helpers: tool(), tool_reject()

Examples

```
# See ?tool() for a full example using this function.
# We're creating a tool around R's `rnorm()` function to allow the chatbot to
# generate random numbers from a normal distribution.
tool_rnorm <- tool(
  rnorm,
  # Describe the tool function to the LLM
  .description = "Drawn numbers from a random normal distribution",
  # Describe the parameters used by the tool function
  n = type_integer("The number of observations. Must be a positive integer."),
  mean = type_number("The mean value of the distribution."),
 sd = type_number("The standard deviation of the distribution. Must be a non-negative number."),
  # Tool annotations optionally provide additional context to the LLM
  .annotations = tool_annotations(
    title = "Draw Random Normal Numbers",
    read_only_hint = TRUE, # the tool does not modify any state
    open_world_hint = FALSE # the tool does not interact with the outside world
  )
)
```

---

tool_reject *Reject a tool call*

Description

Throws an error to reject a tool call. tool_reject() can be used within the tool function to indicate that the tool call should not be processed. tool_reject() can also be called in an Chat$on_tool_request() callback. When used in the callback, the tool call is rejected before the tool function is invoked.

Here's an example where `utils::askYesNo()` is used to ask the user for permission before accessing their current working directory. This happens directly in the tool function and is appropriate when you write the tool definition and know exactly how it will be called.

```
chat <- chat_openai(model = "gpt-4.1-nano")

list_files <- function() {
  allow_read <- utils::askYesNo(
    "Would you like to allow access to your current directory?"
  )
  if (isTRUE(allow_read)) {
    dir(pattern = "[.](r|R|csv)$")
  } else {
    tool_reject()
  }
}

chat$register_tool(tool(
  list_files,
  "List files in the user's current directory"
))

chat$chat("What files are available in my current directory?")
#> [tool call] list_files()
#> Would you like to allow access to your current directory? (Yes/no/cancel) no
#> #> Error: Tool call rejected. The user has chosen to disallow the tool #' call.
#> It seems I am unable to access the files in your current directory right now.
#> If you can tell me what specific files you're looking for or if you can #' provide
#> the list, I can assist you further.

chat$chat("Try again.")
#> [tool call] list_files()
#> Would you like to allow access to your current directory? (Yes/no/cancel) yes
#> #> app.R
#> #> data.csv
#> The files available in your current directory are "app.R" and "data.csv".
```

You can achieve a similar experience with tools written by others by using a `tool_request` callback. In the next example, imagine the tool is provided by a third-party package. This example implements a simple menu to ask the user for consent before running *any* tool.

```
packaged_list_files_tool <- tool(
  function() dir(pattern = "[.](r|R|csv)$"),
  "List files in the user's current directory"
)

chat <- chat_openai(model = "gpt-4.1-nano")
chat$register_tool(packaged_list_files_tool)
```

```
always_allowed <- c()

# ContentToolRequest
chat$on_tool_request(function(request) {
  if (request@name %in% always_allowed) return()

  answer <- utils::menu(
    title = sprintf("Allow tool `%s()` to run?", request@name),
    choices = c("Always", "Once", "No"),
    graphics = FALSE
  )

  if (answer == 1) {
    always_allowed <<- append(always_allowed, request@name)
  } else if (answer %in% c(0, 3)) {
    tool_reject()
  }
})

# Try choosing different answers to the menu each time
chat$chat("What files are available in my current directory?")
chat$chat("How about now?")
chat$chat("And again now?")
```

Usage

```
tool_reject(reason = "The user has chosen to disallow the tool call.")
```

Arguments

`reason`	A character string describing the reason for rejecting the tool call.

Value

Throws an error of class `ellmer_tool_reject` with the provided reason.

See Also

Other tool calling helpers: `tool()`, `tool_annotations()`

Turn *A user, assistant, or system turn*

Description

Every conversation with a chatbot consists of pairs of user and assistant turns, corresponding to an HTTP request and response. These turns are represented by the `Turn` object, which contains a list of Contents representing the individual messages within the turn. These might be text, images, tool requests (assistant only), or tool responses (user only).

`UserTurn`, `AssistantTurn`, and `SystemTurn` are specialized subclasses of `Turn` for different types of conversation turns. `AssistantTurn` includes additional metadata about the API response.

Note that a call to `$chat()` and related functions may result in multiple user-assistant turn cycles. For example, if you have registered tools, ellmer will automatically handle the tool calling loop, which may result in any number of additional cycles. Learn more about tool calling in `vignette("tool-calling")`.

Usage

```
Turn(role = NULL, contents = list(), tokens = NULL)

UserTurn(contents = list())

SystemTurn(contents = list())

AssistantTurn(
  contents = list(),
  json = list(),
  tokens = c(NA_real_, NA_real_, NA_real_),
  cost = NA_real_,
  duration = NA_real_
)

AssistantPartialTurn(
  contents = list(),
  json = list(),
  tokens = c(NA_real_, NA_real_, NA_real_),
  cost = NA_real_,
  duration = NA_real_,
  reason = "interrupted"
)
```

Arguments

`role`	**[Deprecated]** For system, user and assistant turns, use `SystemTurn()`, `UserTurn()`, and `AssistantTurn()`, respectively.
`contents`	A list of Content objects.
`tokens`	A numeric vector of length 3 representing the number of input tokens (uncached), output tokens, and input tokens (cached) used in this turn.
`json`	The serialized JSON corresponding to the underlying data of the turns. This is useful if there's information returned by the provider that ellmer doesn't otherwise expose.

`cost`	The cost of the turn in dollars.
`duration`	The duration of the request in seconds.
`reason`	A character string describing why the turn was interrupted. Defaults to `"interrupted"`.

Value

An S7 `Turn` object

An S7 `AssistantTurn` object

An S7 `AssistantPartialTurn` object

Examples

```
UserTurn(list(ContentText("Hello, world!")))
```

Type *Type definitions for function calling and structured data extraction.*

Description

These S7 classes are provided for use by package devlopers who are extending ellmer. In every day use, use `type_boolean()` and friends.

Usage

```
TypeBasic(description = NULL, required = TRUE, type = stop("Required"))

TypeEnum(description = NULL, required = TRUE, values = character(0))

TypeArray(description = NULL, required = TRUE, items = Type())

TypeJsonSchema(description = NULL, required = TRUE, json = list())

TypeIgnore(description = NULL, required = TRUE)

TypeObject(
  description = NULL,
  required = TRUE,
  properties = list(),
  additional_properties = FALSE
)
```

Arguments

description	The purpose of the component. This is used by the LLM to determine what values to pass to the tool or what values to extract in the structured data, so the more detail that you can provide here, the better.
required	Is the component or argument required? In type descriptions for structured data, if required = FALSE and the component does not exist in the data, the LLM may hallucinate a value. Only applies when the element is nested inside of a type_object(). In tool definitions, required = TRUE signals that the LLM should always provide a value. Arguments with required = FALSE should have a default value in the tool function's definition. If the LLM does not provide a value, the default value will be used.
type	Basic type name. Must be one of boolean, integer, number, or string.
values	Character vector of permitted values.
items	The type of the array items. Can be created by any of the type_ function.
json	A JSON schema object as a list.
properties	Named list of properties stored inside the object. Each element should be an S7 Type object.‘
additional_properties	Can the object have arbitrary additional properties that are not explicitly listed? Only supported by Claude.

Value

S7 objects inheriting from Type

Examples

```
TypeBasic(type = "boolean")
TypeArray(items = TypeBasic(type = "boolean"))
```

---

type_boolean *Type specifications*

Description

These functions specify object types in a way that chatbots understand and are used for tool calling and structured data extraction. Their names are based on the JSON schema, which is what the APIs expect behind the scenes. The translation from R concepts to these types is fairly straightforward.

- type_boolean(), type_integer(), type_number(), and type_string() each represent scalars. These are equivalent to length-1 logical, integer, double, and character vectors (respectively).
- type_enum() is equivalent to a length-1 factor; it is a string that can only take the specified values.

- `type_array()` is equivalent to a vector in R. You can use it to represent an atomic vector: e.g. `type_array(type_boolean())` is equivalent to a logical vector and `type_array(type_string())` is equivalent to a character vector). You can also use it to represent a list of more complicated types where every element is the same type (R has no base equivalent to this), e.g. `type_array(type_array(type_string()))` represents a list of character vectors.
- `type_object()` is equivalent to a named list in R, but where every element must have the specified type. For example, `type_object(a = type_string(), b = type_array(type_integer()))` is equivalent to a list with an element called `a` that is a string and an element called `b` that is an integer vector.
- `type_ignore()` is used in tool calling to indicate that an argument should not be provided by the LLM. This is useful when the R function has a default value for the argument and you don't want the LLM to supply it.
- `type_from_schema()` allows you to specify the full schema that you want to get back from the LLM as a JSON schema. This is useful if you have a pre-defined schema that you want to use directly without manually creating the type using the `type_*()` functions. You can point to a file with the `path` argument or provide a JSON string with `text`. The schema must be a valid JSON schema object.

Usage

```
type_boolean(description = NULL, required = TRUE)

type_integer(description = NULL, required = TRUE)

type_number(description = NULL, required = TRUE)

type_string(description = NULL, required = TRUE)

type_enum(values, description = NULL, required = TRUE)

type_array(items, description = NULL, required = TRUE)

type_object(
  .description = NULL,
  ...,
  .required = TRUE,
  .additional_properties = FALSE
)

type_from_schema(text, path)

type_ignore()
```

Arguments

`description`, `.description`
: The purpose of the component. This is used by the LLM to determine what values to pass to the tool or what values to extract in the structured data, so the more detail that you can provide here, the better.

`required`, `.required`
: Is the component or argument required?

  In type descriptions for structured data, if `required = FALSE` and the component does not exist in the data, the LLM may hallucinate a value. Only applies when the element is nested inside of a `type_object()`.

  In tool definitions, `required = TRUE` signals that the LLM should always provide a value. Arguments with `required = FALSE` should have a default value in the tool function's definition. If the LLM does not provide a value, the default value will be used.

`values`
: Character vector of permitted values.

`items`
: The type of the array items. Can be created by any of the `type_` function.

`...`
: <dynamic-dots> Name-type pairs defining the components that the object must possess.

`.additional_properties`
: Can the object have arbitrary additional properties that are not explicitly listed? Only supported by Claude.

`text`
: A JSON string.

`path`
: A file path to a JSON file.

Examples

```
# An integer vector
type_array(type_integer())

# The closest equivalent to a data frame is an array of objects
type_array(type_object(
   x = type_boolean(),
   y = type_string(),
   z = type_number()
))

# There's no specific type for dates, but you use a string with the
# requested format in the description (it's not guaranteed that you'll
# get this format back, but you should most of the time)
type_string("The creation date, in YYYY-MM-DD format.")
type_string("The update date, in dd/mm/yyyy format.")
```

Index